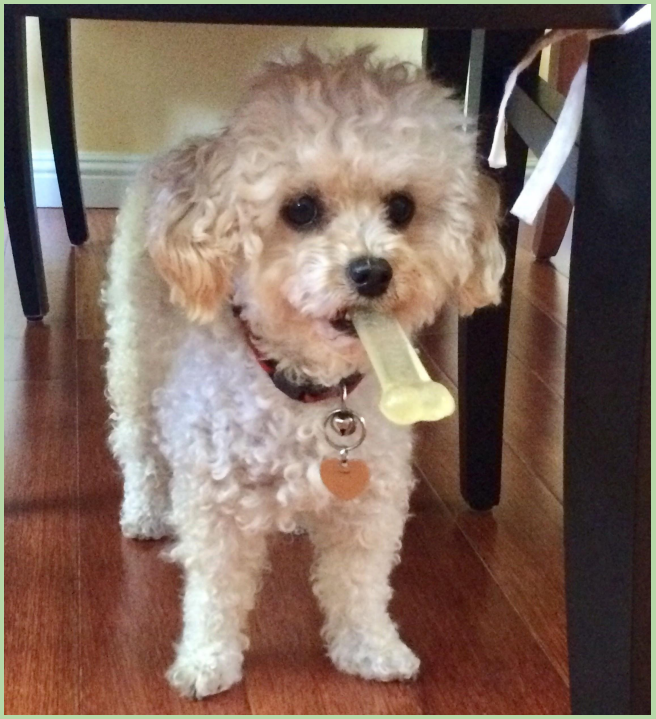
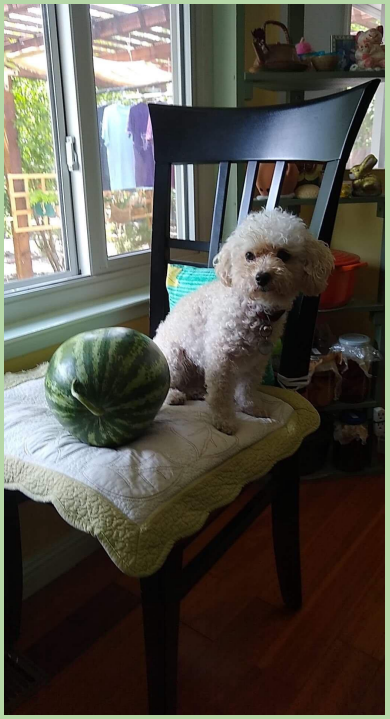
Lecture 10
Object Oriented Programming and Modeling Project



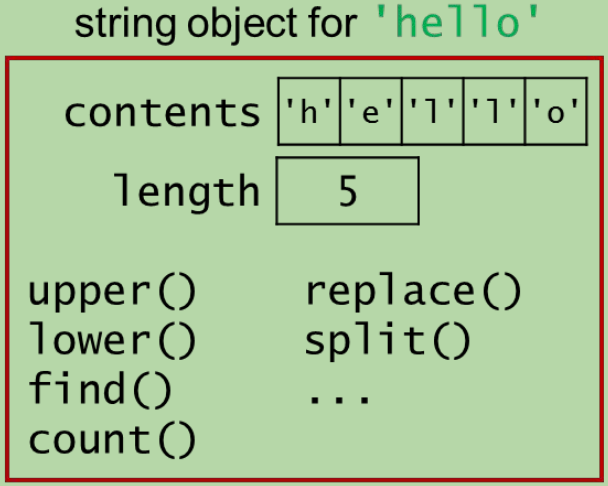


UTA Joseph's dog Ponyo with various objects. Very object oriented!

Classes:
Defining New Types of Objects

Objects, Objects, Everywhere!

- *Recall:* Strings are objects with:
  - *attributes* – data values inside the object
  - *methods* – functions inside the object



- In fact, *everything* in Python is an object!
  - integers
  - floats
  - lists
  - booleans
  - file handles
  - functions
  - ...

Classes

- A *class* is a blueprint – a definition of a data type.
  - specifies the attributes and methods of that type

- Objects are built according to the blueprint provided by their class.
  - they are "values" aka ***instances*** of that type
  - use the type function to determine the class:

```
>>> type(111)
<class 'int'>

>>> type(3.14159)
<class 'float'>

>>> type('hello!')
<class 'str'>

>>> type([1, 2, 3])
<class 'list'>
```

Another Analogy

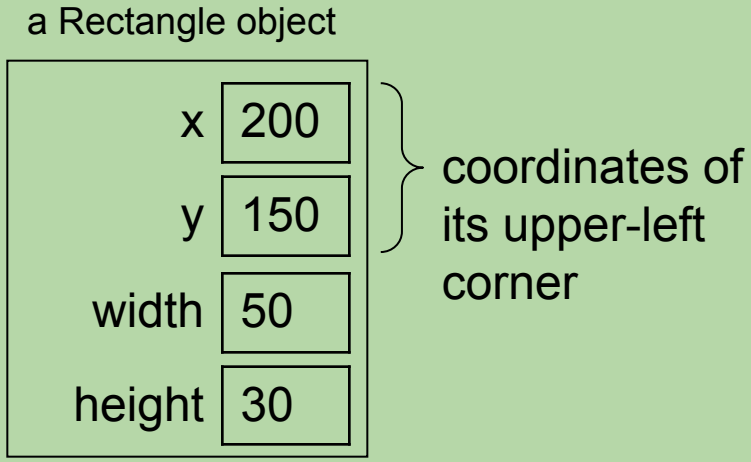
- A class is like a cookie cutter.
  - specifies the "shape" that all objects of that type should have

- Objects are like the cookies.
  - created with the "shape" specified by their class

Creating Your Own Classes

- In an *object-oriented* programming language, you can define your own classes.
  - your own types of objects
  - your own data types!

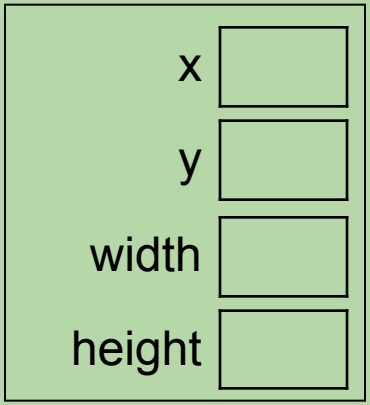
- Example: let's say that we want objects that represent rectangles.



- A Rectangle object could have methods for:
  - computing its area, perimeter, etc.
  - growing it (changing its dimensions), moving it, etc.

An Initial Rectangle Class

```
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """

    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```



- __init__ is the ***constructor.***
  - it's used to create new objects
  - it specifies the attributes

- Inside its methods, an object refers to itself as self!

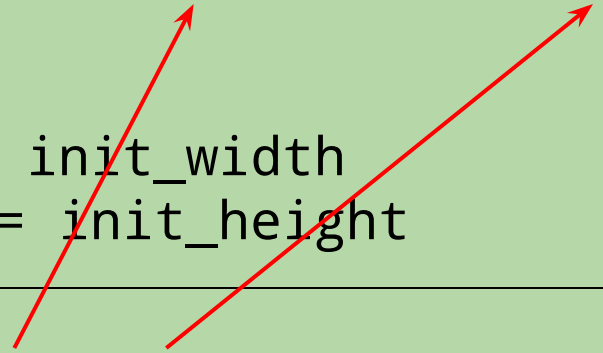
Constructing and Using an Object

```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

```
>>> r1 = Rectangle(100, 50)    # calls __init__!
```

Constructing and Using an Object

```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

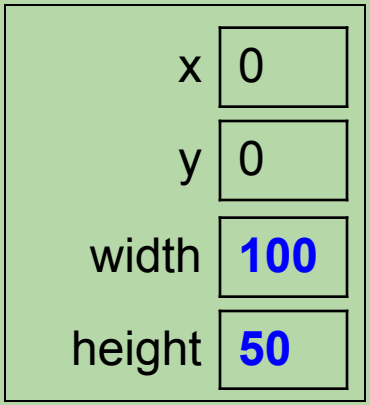


```
>>> r1 = Rectangle(100, 50)    # calls __init__!
```

Constructing and Using an Object

```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

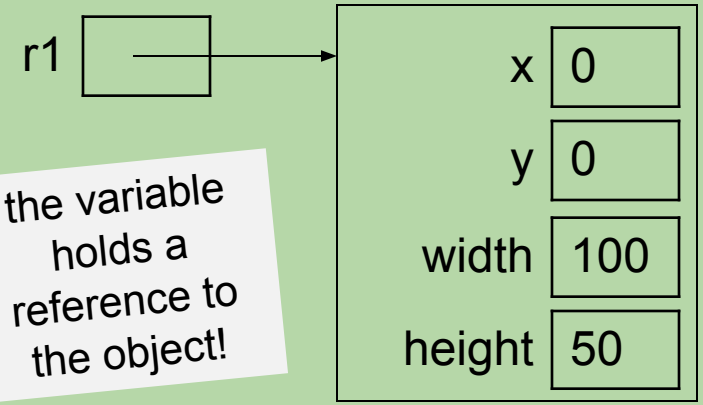
```
>>> r1 = Rectangle(100, 50)    # calls __init__!
```



Constructing and Using an Object

```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

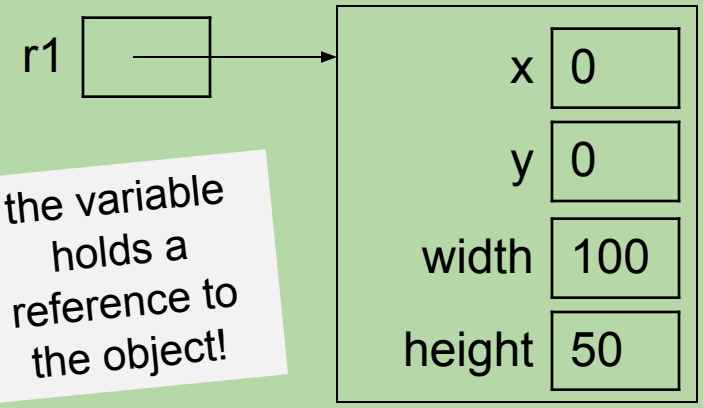
```
>>> r1 = Rectangle(100, 50)    # calls __init__!
```



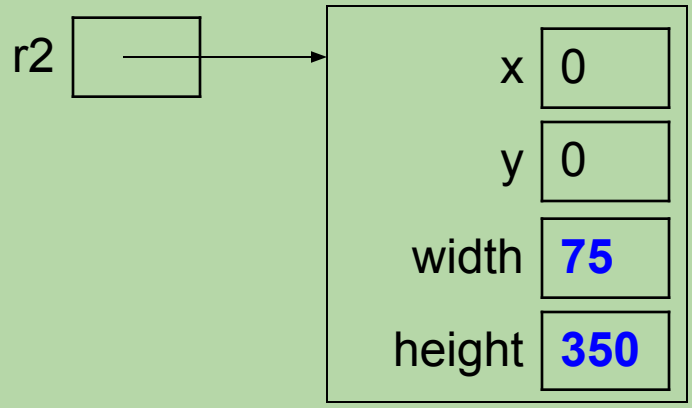
Constructing and Using an Object

```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height
```

```
>>> r1 = Rectangle(100, 50)    # calls __init__!
>>> r2 = Rectangle(75, 350)    # construct another one!
```

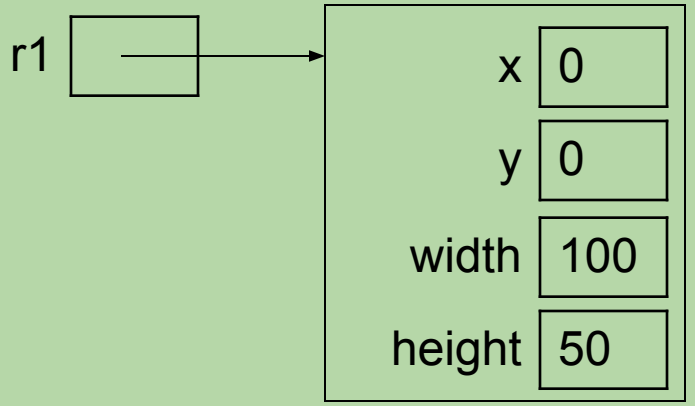


the variable holds a reference to the object!



Accessing and Modifying an Object's Attributes

```
>>> r1 = Rectangle(100, 50)
```

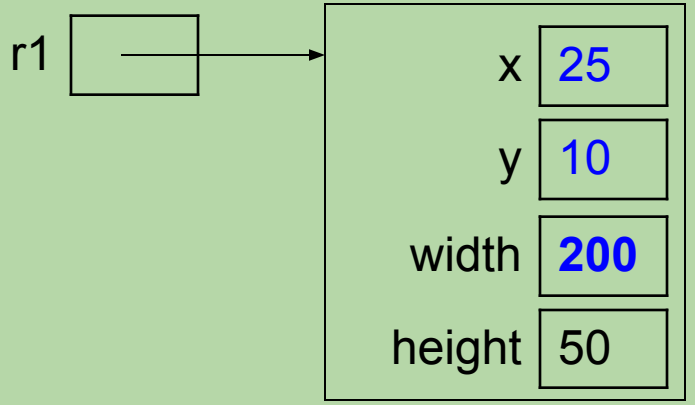


- Access the attributes using *dot notation:*

```
>>> r1.width
100
>>> r1.height
50
```

- Modify them as you would any other variable:

```
>>> r1.x = 25
>>> r1.y = 10
>>> r1.width *= 2
```



Application Programs

- Our Rectangle class is *not* a program.

- Instead, it will be used by code defined elsewhere.
  - referred to as ***Application programs*** or ***Application code***
  - referred to as ***Client programs*** or ***Client code***

- More generally, when we define a new type of object,
  we create a building block that can be used in other code.
  - just like the objects from the built-in classes:
    str, list, int, etc.
  - our programs have been *applications* that use those classes!
  - our programs have been *clients* of those classes!

Initial Application Program

```
# construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
area1 = r1.width * r1.height
print('area =', area1)

print('r2:', r2.width, 'x', r2.height)
area2 = r2.width * r2.height
print('area =', area2)

# grow both Rectangles
r1.width += 50
r1.height += 10
r2.width += 5
r2.height += 30

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

Using Methods to Capture an Object's Behavior

- Rather than having the Application grow the Rectangle objects, we'd like to give each Rectangle object the ability to grow itself.

- We do so by adding a method to the class:

```
class Rectangle:
    """ the Rectangle constructor """
    def __init__(self, init_width, init_height):
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

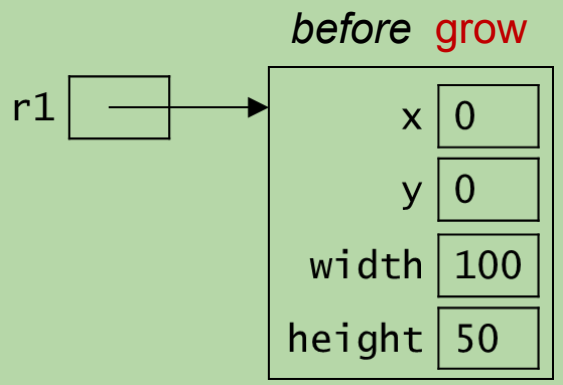
note that the first “parameter” for a class method is always ‘self’

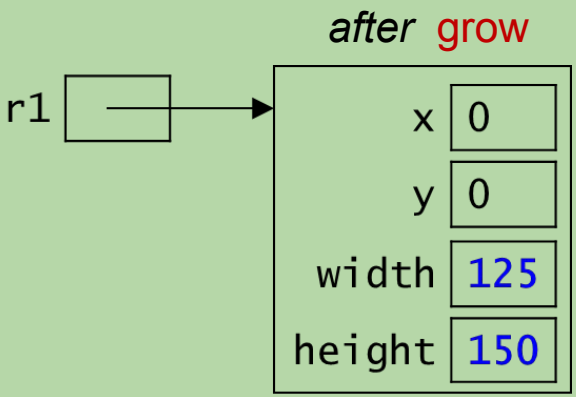
when we call this method, the self parameter is implicit: r.grow(dwidth, dheight)

Calling a Method

```
class Rectangle:
    ...
    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

```
>>> r1 = Rectangle(100, 50)

>>> r1.grow(25, 100)

>>> r1.width
125

>>> r1.height
150
```





Another Example of a Method

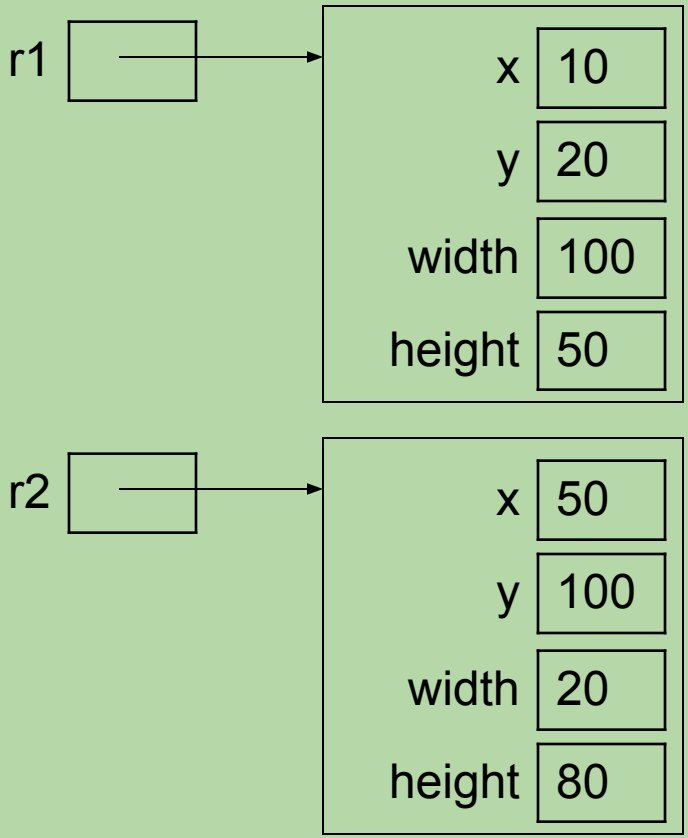
- Here's a method for getting the area of a Rectangle:

```
def area(self):
    return self.width * self.height
```

- Sample method calls:

```
>>> r1.area()
5000
>>> r2.area()
1600
```

  - we're asking r1 and r2 to give us their areas
  - nothing in the parentheses because the necessary info. is in the objects' attributes!



Second Version of our Rectangle Class

```
# assume this is in rectangle.py

class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """
    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight


    def area(self):
            return self.width * self.height
```

Original Application Program...

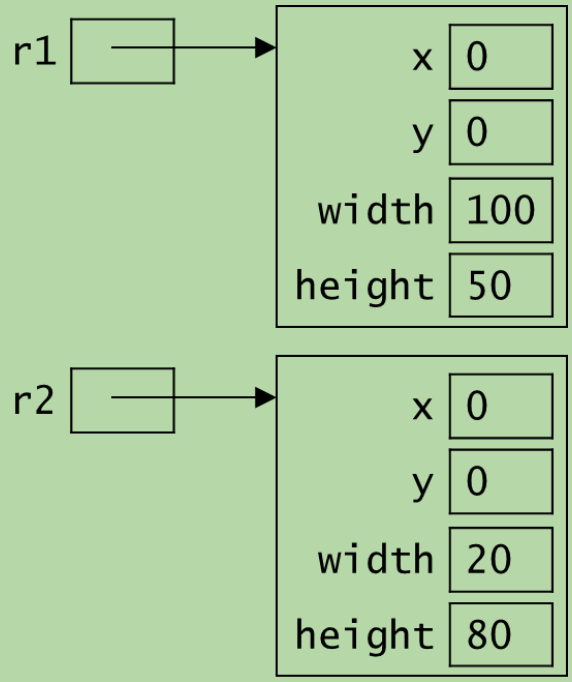
```
from rectangle import *

# construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
area1 = r1.width * r1.height
print('area =', area1)

print('r2:', r2.width, 'x', r2.height)
area2 = r2.width * r2.height
print('area =', area2)

# grow both Rectangles
r1.width += 50
r1.height += 10
r2.width += 5
r2.height += 30

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

Simplified Application Program

```
from rectangle import *

# construct two Rectangle objects
r1 = Rectangle(100, 50)
r2 = Rectangle(75, 350)

# print dimensions and area of each
print('r1:', r1.width, 'x', r1.height)
print('area =', r1.area())

print('r2:', r2.width, 'x', r2.height)
print('area =', r2.area())

# grow both Rectangles
r1.grow(50, 10)
r2.grow(5, 30)

# print new dimensions
print('r1:', r1.width, 'x', r1.height)
print('r2:', r2.width, 'x', r2.height)
```

Be Objective!

```
class Rectangle:
  ...
   def grow(self, dwidth, dheight):
      ...

   def area(self):
            ...
```

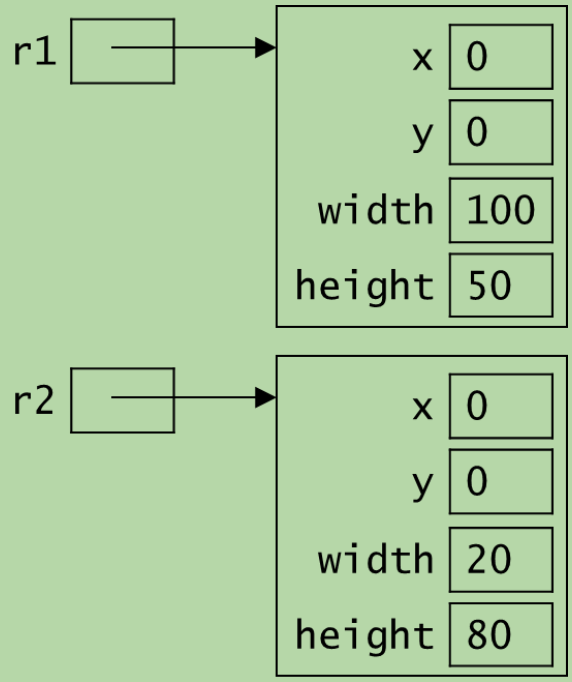



```
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)
```

- Give an expression for:
  - the width of r1:
  - the height of r2:
- Write an assignment that changes r1's x-coordinate to 50:
- Write a method call that:
  - increases r2's width by 5 and height by 10:
  - gets r1's area:

Be Objective!

```
class Rectangle:
  ...
   def grow(self, dwidth, dheight):
      ...

   def area(self):
            ...
```




```
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)
```

- Give an expression for:
  - the width of r1: r1.width
  - the height of r2: r2.height
- Write an assignment that changes r1's x-coordinate to 50:
  r1.x = 50
- Write a method call that:
  - increases r2's width by 5 and height by 10: r2.grow(5, 10)
  - gets r1's area: r1.area()

Method vs. Function

- Our area ***method*** is part of the Rectangle class:

```
class Rectangle:
    ...
    def area(self):       # methods have a self
        return self.width * self.height
```

  - thus, it is inside Rectangle objects
  - sample call:

```
r.area()
```

- Here's a ***function*** that takes two Rectangle objects as inputs:

```
def total_area(r1, r2):   # functions don't
    return r1.area() + r2.area()
```

  - it is *not* part of the class and is *not* inside Rectangle objects
  - sample call:

```
total_area(r, other_r)
```

  - it is a client of the Rectangle class!

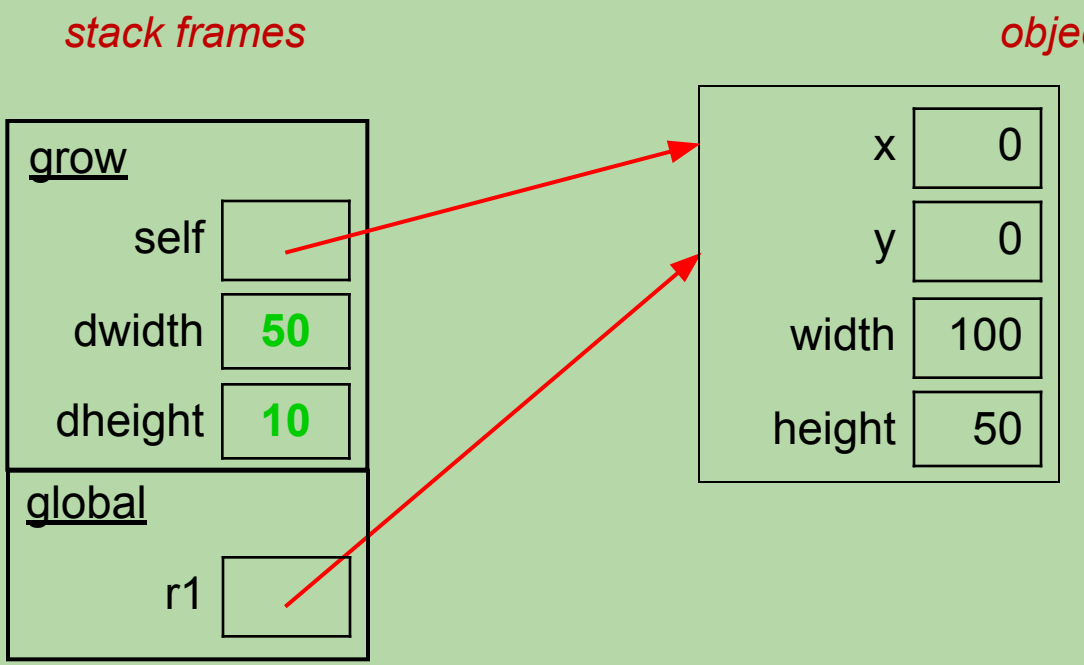
Methods That Modify an Object

```
class Rectangle:
    """ a blueprint for objects that represent
        a rectangular shape
    """
    def __init__(self, init_width, init_height):
        """ the Rectangle constructor """
        self.x = 0
        self.y = 0
        self.width = init_width
        self.height = init_height

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
        # why don't we need a return?

    def area(self):
            return self.width * self.height
```

Methods That Modify an Object

```
r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)
```

```
def grow(self, dwidth, dheight):
    self.width += dwidth
    self.height += dheight
```

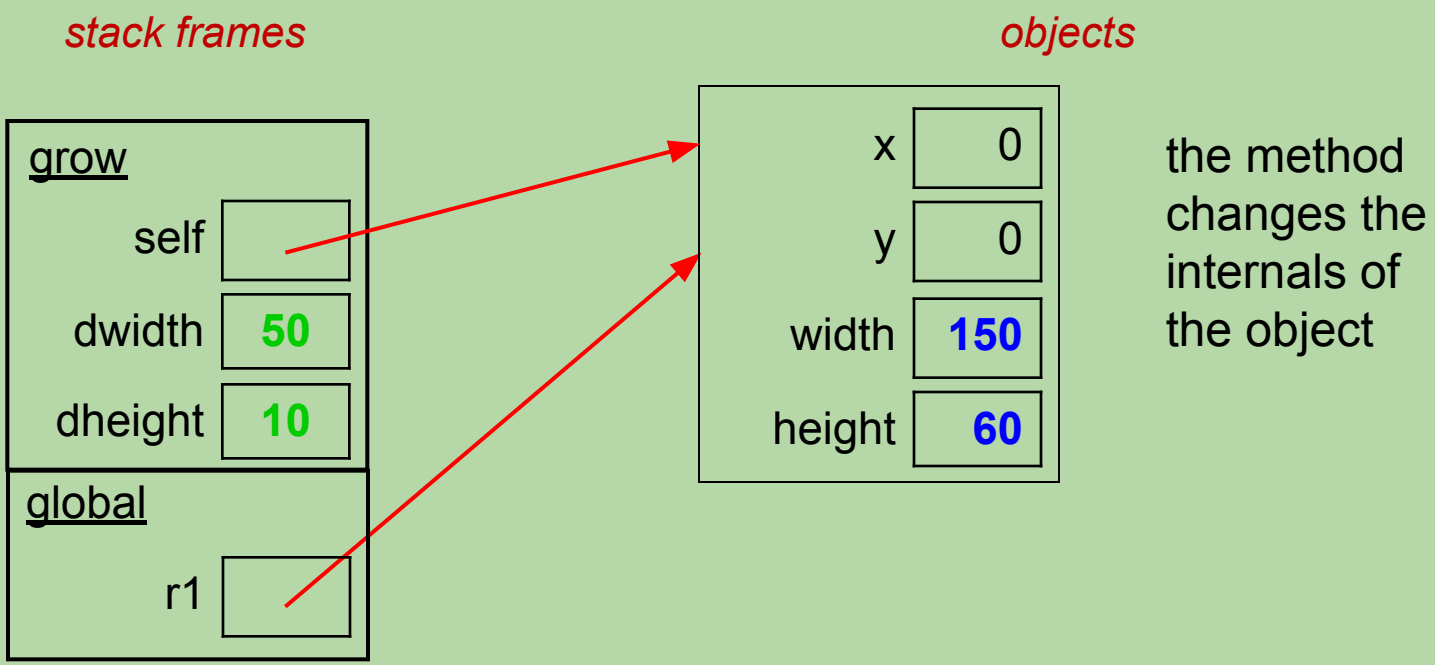


self refers to the object on which the method is called

Methods That Modify an Object

```
r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)
```

```
def grow(self, dwidth, dheight):
    self.width += dwidth
    self.height += dheight
```

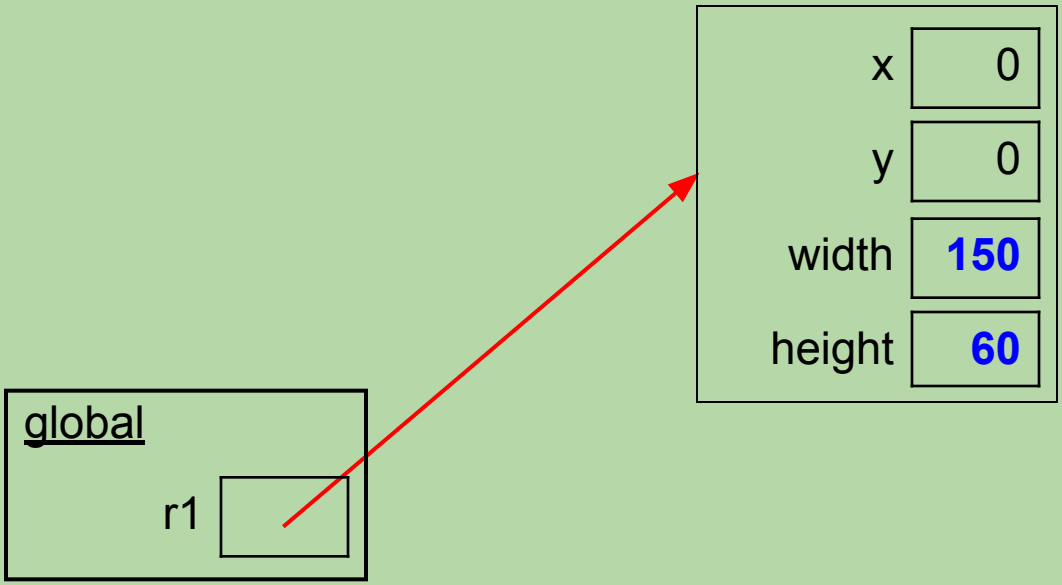


Methods That Modify an Object

```
r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)
```

```
def grow(self, dwidth, dheight):
    self.width += dwidth
    self.height += dheight
```

stack frames *objects*

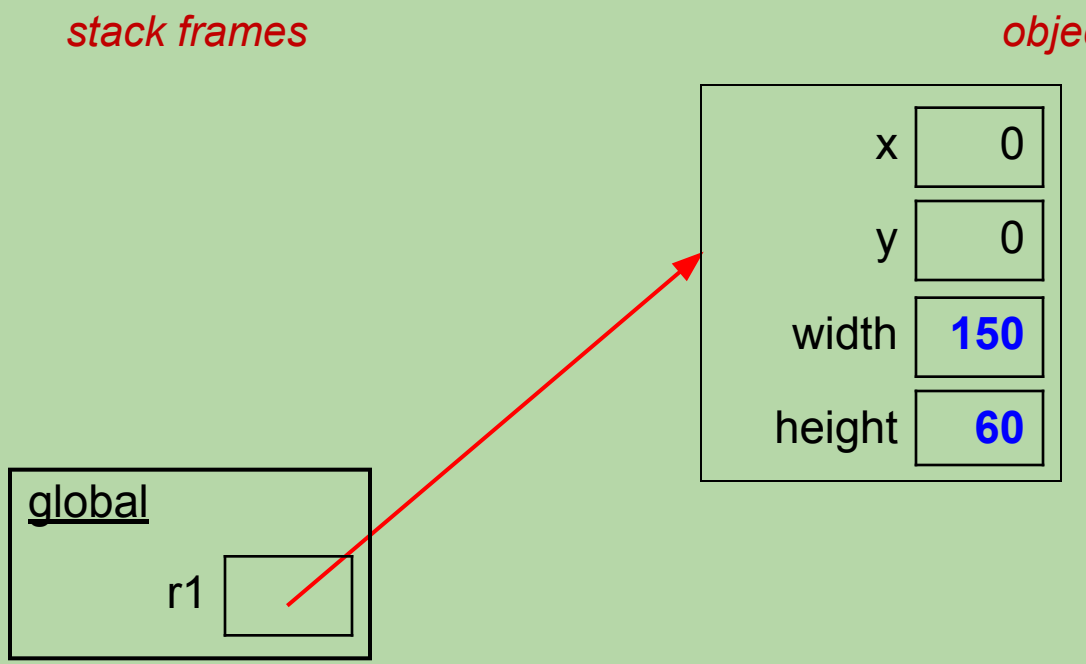


those changes are still there after the method returns

Methods That Modify an Object

```
r1 = Rectangle(100, 50)
r1.grow(50, 10)
print('r1:', r1.width, 'x', r1.height)
```

output: r1: 150 x 60



Which of these is a correct perimeter method?

A.

```
def perimeter(self, width, height):
    return 2*width + 2*height
```

B.

```
def perimeter():
    return 2*self.width + 2*self.height
```

C.

```
def perimeter(self):
    return 2*self.width + 2*self.height
```

D. more than one of these

E. none of these

Which of these is a correct perimeter method?

A.

```
def perimeter(self, width, height):
    return 2*width + 2*height
```

B.

```
def perimeter():
    return 2*self.width + 2*self.height
```

C.

```
def perimeter(self):
    return 2*self.width + 2*self.height
```

D. more than one of these

E. none of these

Fill in the blank to call the perimeter method.

```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height
```

```
r = Rectangle(35, 20)

perim = _____________________________
```

A. perimeter(r)

B. perimeter(self, r)

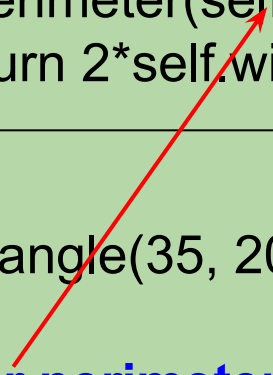
C. perimeter(self, 35, 20)

D. r.perimeter(35, 20)

E. r.perimeter()

Fill in the blank to call the perimeter method.

```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height
```



```
r = Rectangle(35, 20)

perim = r.perimeter()
```

A. perimeter(r)

B. perimeter(self, r)

C. perimeter(self, 35, 20)

D. r.perimeter(35, 20)

E. **r.perimeter()**

scale Method

```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height

    def scale(_____________________________):
```

Write a method called scale that will scale the dimensions of a Rectangle by a specified factor.

sample call:
r.scale(5)

scale Method

```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height

    def scale(self, factor):
        self.width *= factor
        self.height *= factor
```

scale Method

```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height

    def scale(self, factor):
        self.width *= factor
        self.height *= factor
```

```
r = Rectangle(35, 20)
perim = r.perimeter()

# How would we triple the dimensions of r?
```

scale Method

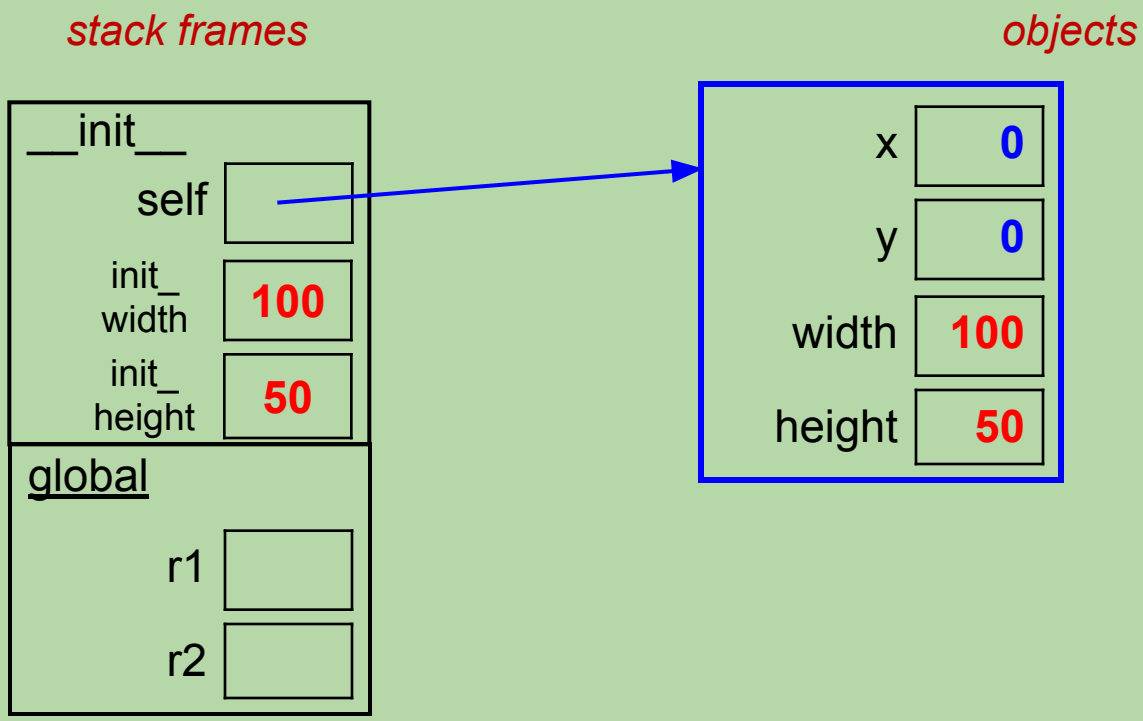
```
class Rectangle:
    ...
    def perimeter(self):
        return 2*self.width + 2*self.height

    def scale(self, factor):
        self.width *= factor
        self.height *= factor
```

```
r = Rectangle(35, 20)
perim = r.perimeter()

# How would we triple the dimensions of r?
r.scale(3)
```

Why doesn't scale *need to return anything?*

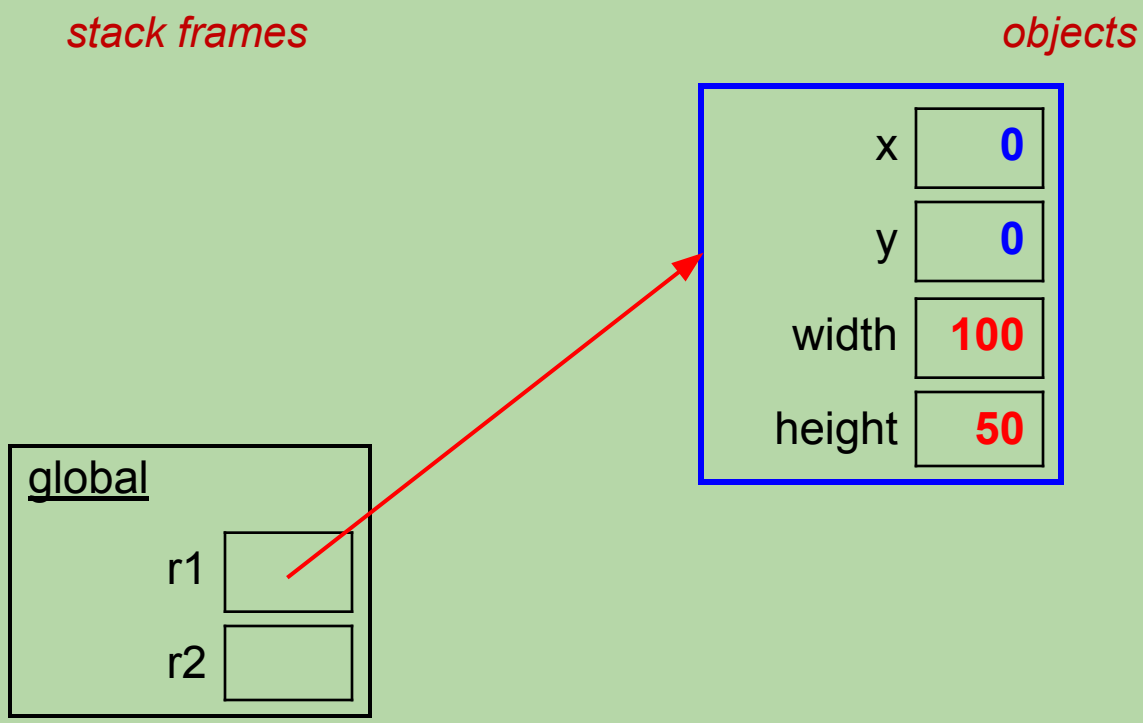
Memory Diagrams for Method Calls

```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



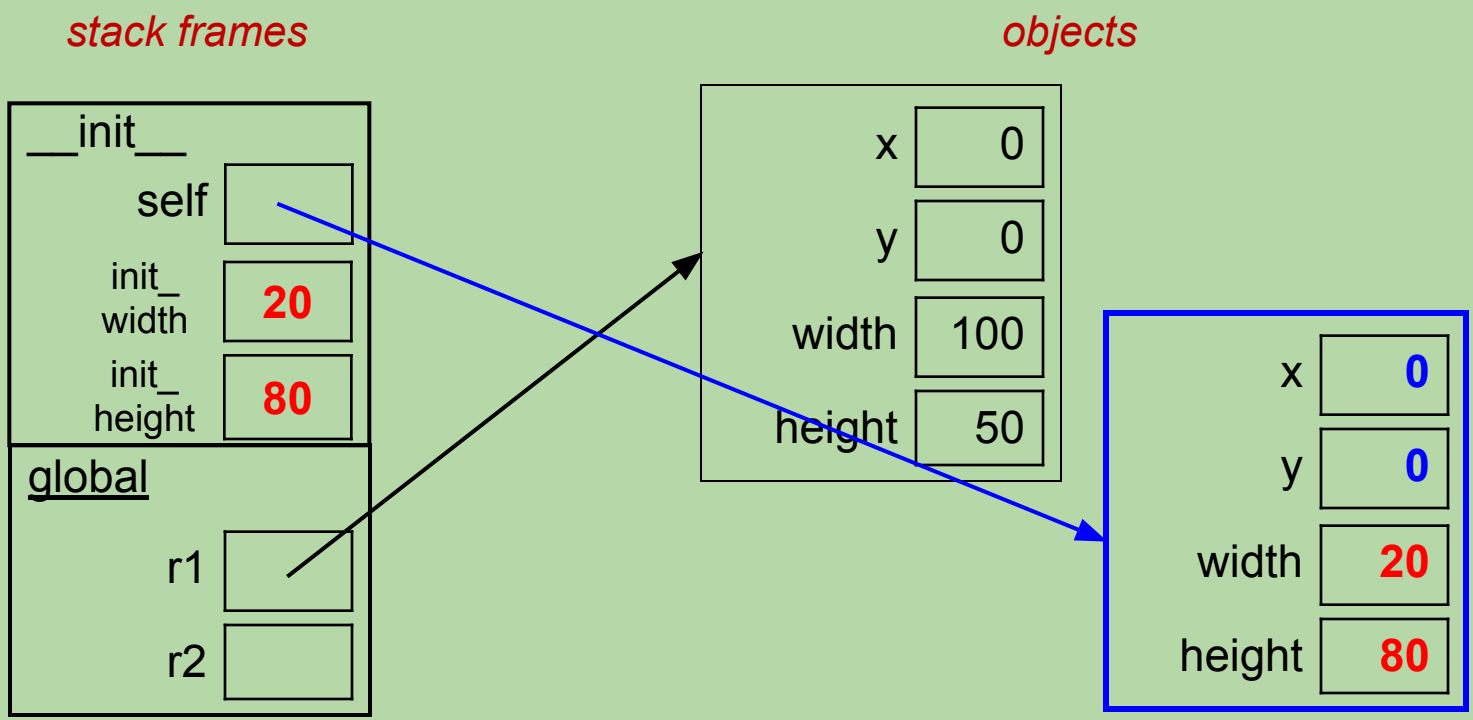
Memory Diagrams for Method Calls

```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



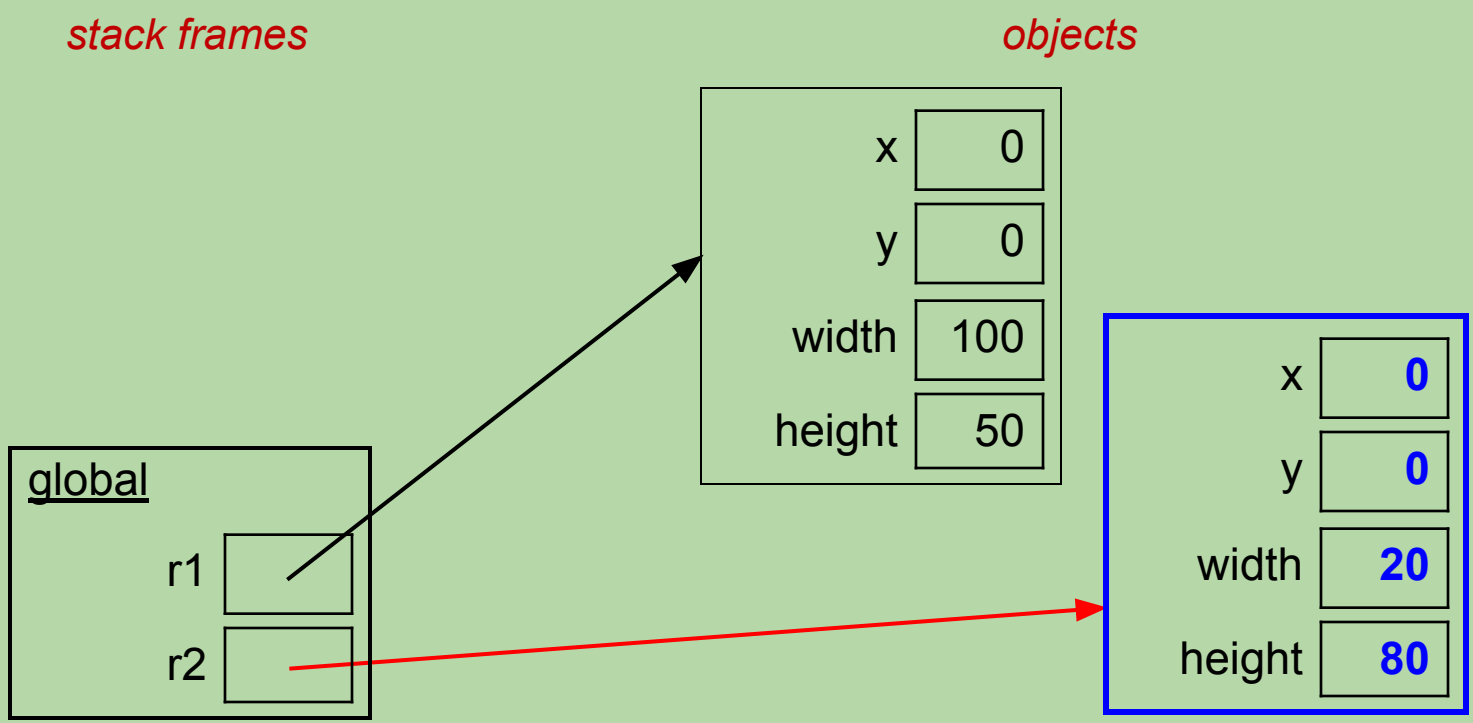
Memory Diagrams for Method Calls

```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



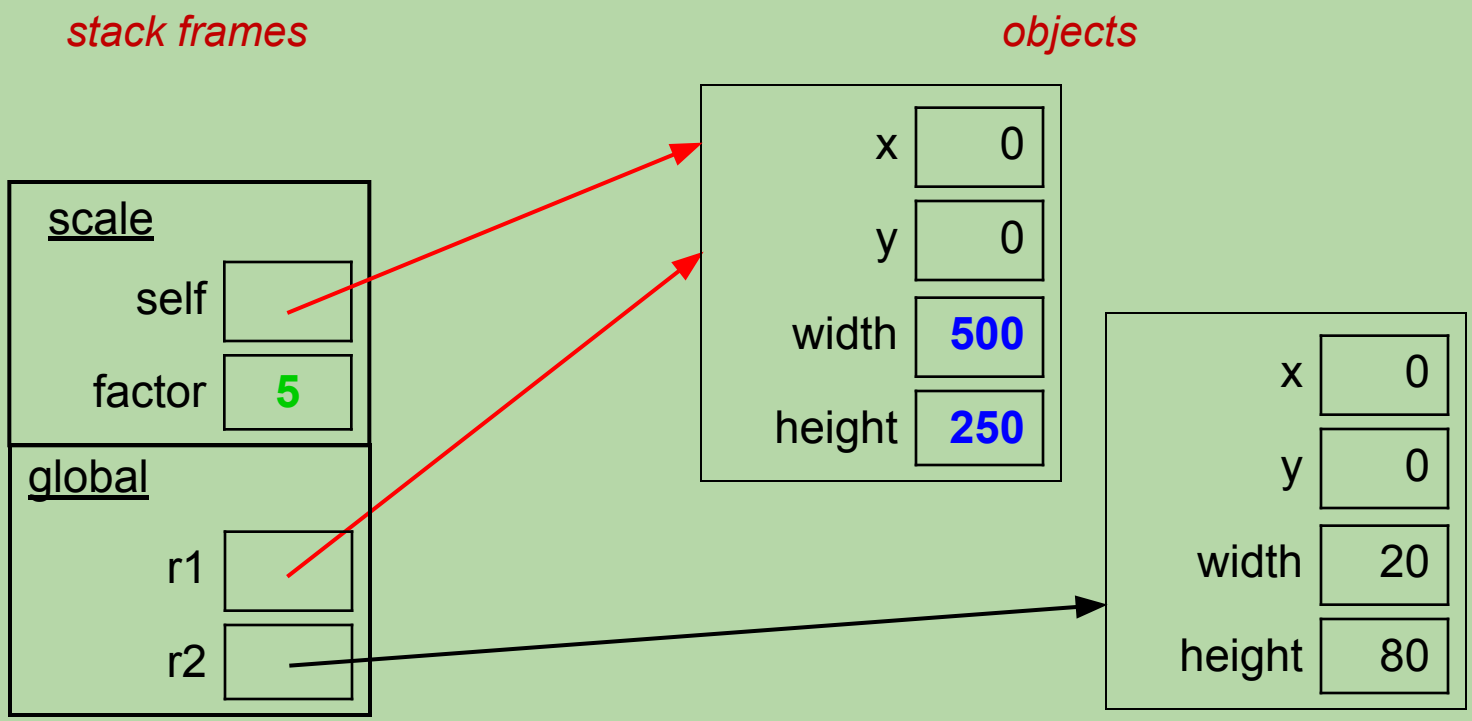
Memory Diagrams for Method Calls

```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



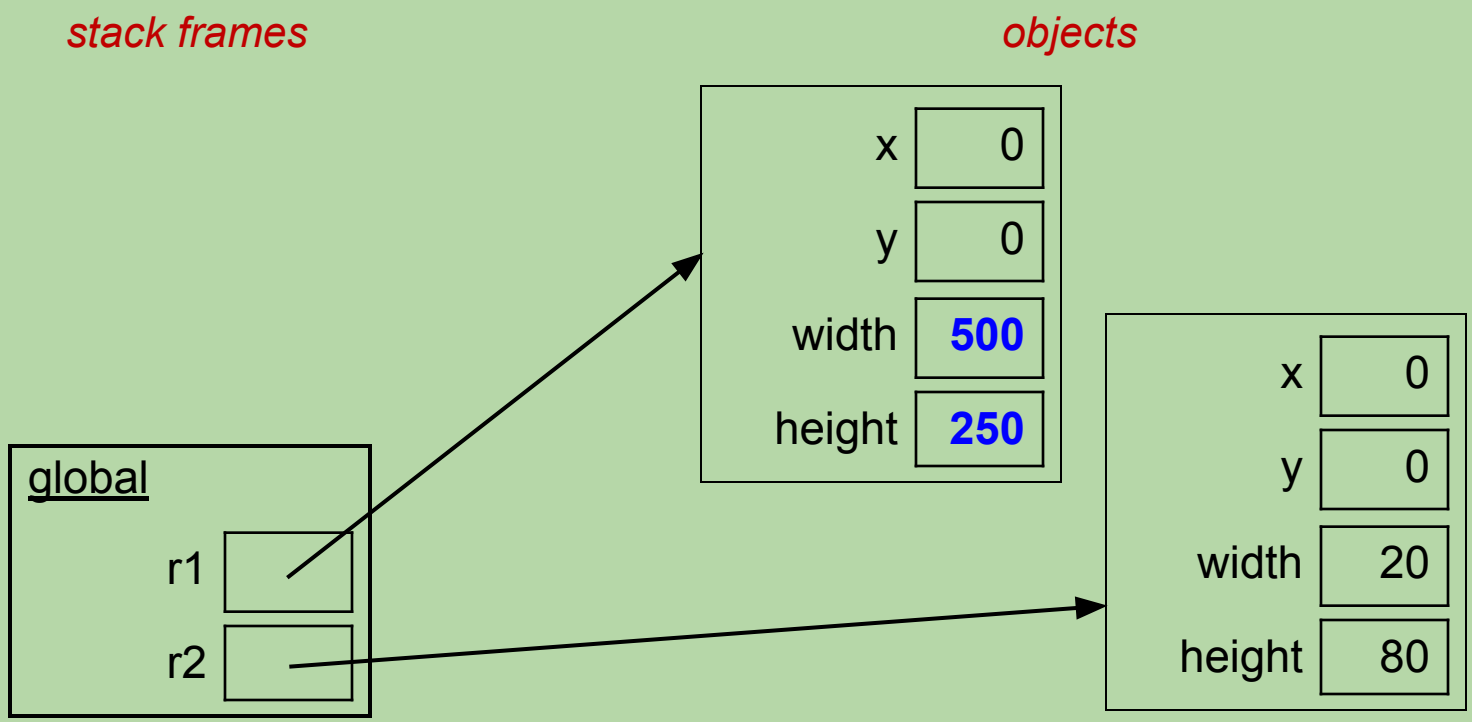
Memory Diagrams for Method Calls

```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



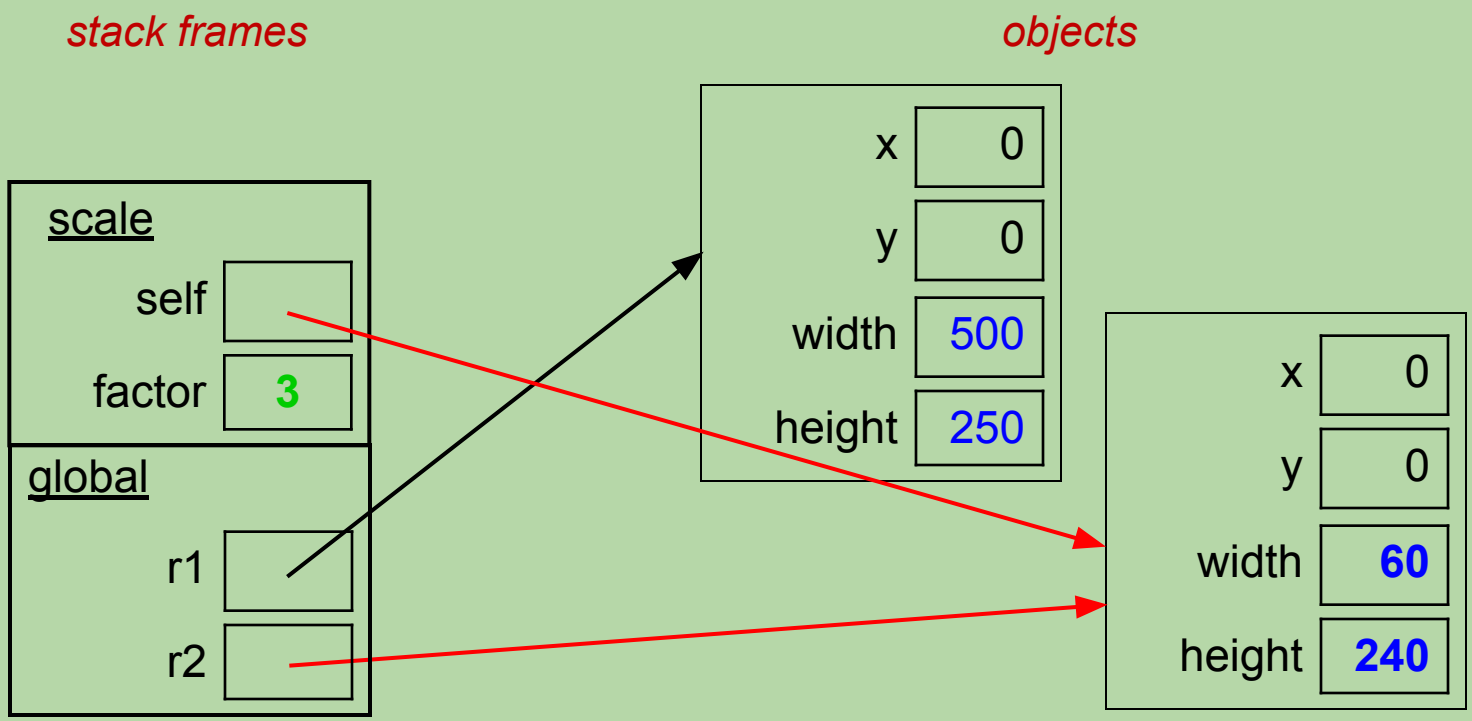
Memory Diagrams for Method Calls

```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



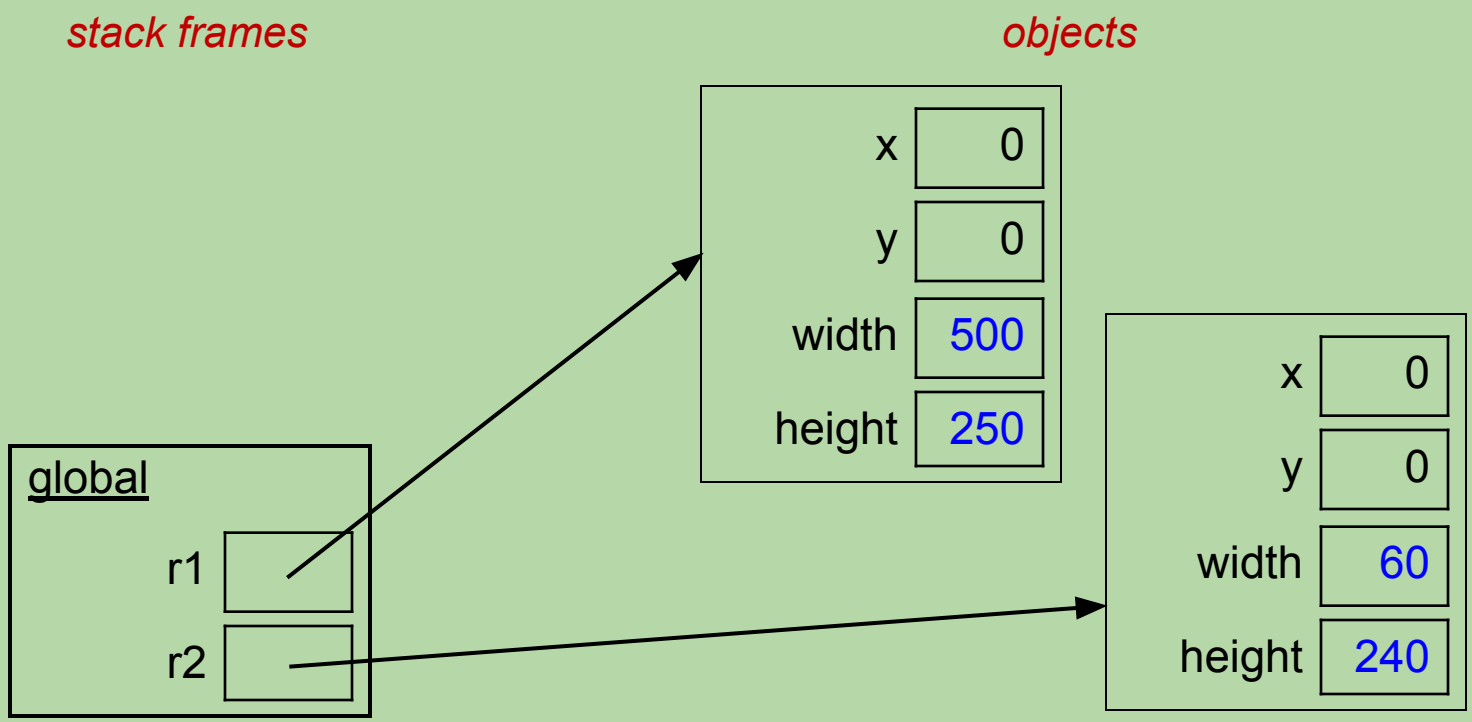
Memory Diagrams for Method Calls

```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



Memory Diagrams for Method Calls

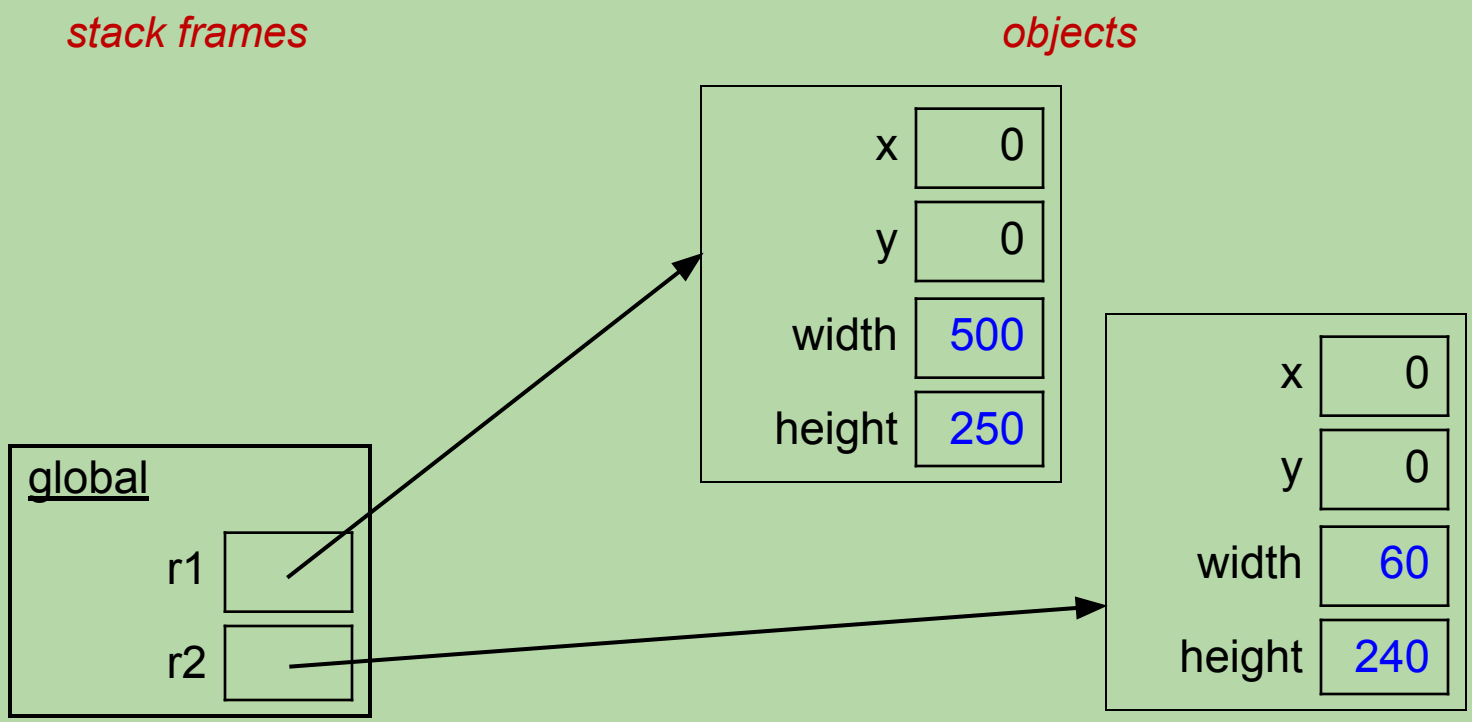
```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```



Memory Diagrams for Method Calls

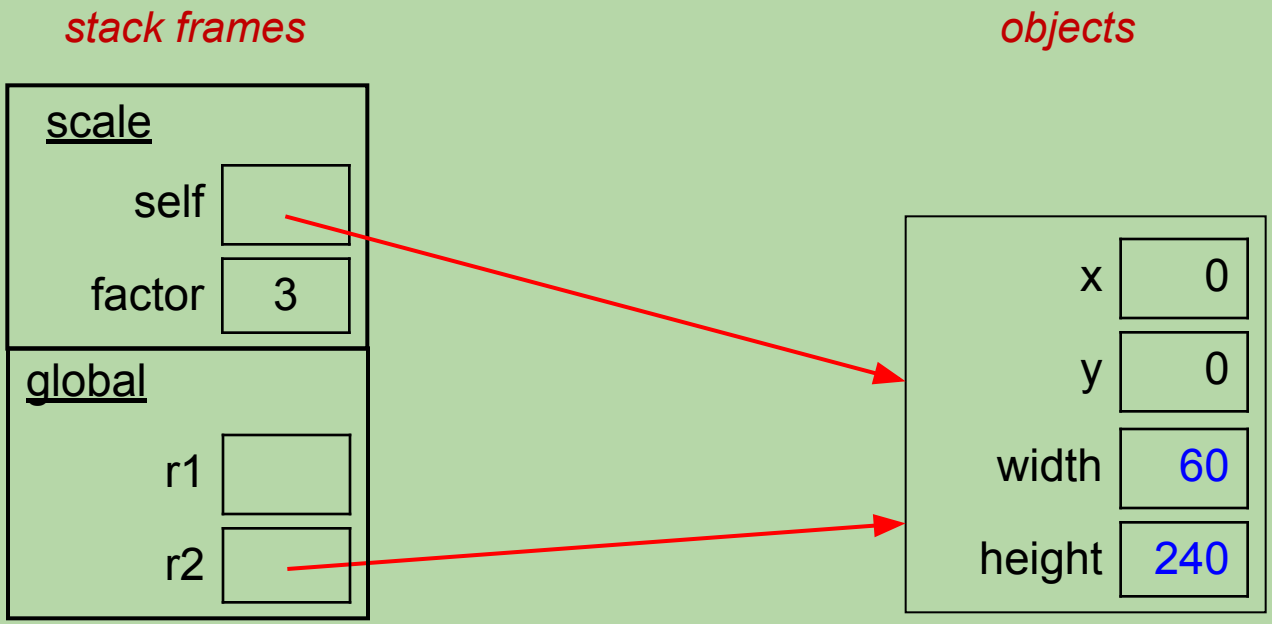
```
# Rectangle Application code
r1 = Rectangle(100, 50)
r2 = Rectangle(20, 80)

r1.scale(5)
r2.scale(3)
print(r1.width, r1.height, r2.width, r2.height)
```

output: 500 250 60 240



No Return Value Is Needed After a Change

- A method operates directly on the called object,
  so any changes it makes will be there after the method returns.
  - example: the call r2.scale(3) from the last slide



- scale gets a copy of the *reference* in r2
- thus, scale's changes can be "seen" using r2
  after scale returns

Project 2
Modeling

UTA Annie honestly could've been a baby model

Text Processing: Stemming

- word → *stem*/*root* of the word
- Examples:

```
stem('love')      ⇒  'lov'
stem('loving')    ⇒  'lov'

stem('stems')     ⇒  'stem'
stem('stemming')  ⇒  'stem'
stem('stem')      ⇒  'stem'

stem('party')     ⇒  'parti'
stem('parties')   ⇒  'parti'
```

Which Word(s) Does It "Get Wrong"?

```
def stem(word):
    if word[-3:] == 'ing':
        word = word[:-3]
    elif word[-2:] == 'er':
        word = word[:-3]
    elif:
        # lots more cases!
        ...

    return word
```

A. `playing`

B. `stemming`

C. `spammer`

D. `reader`

E. more than one (which ones?)

Which Word(s) Does It "Get Wrong"?

```
def stem(word):
    if word[-3:] == 'ing':
        word = word[:-3]
    elif word[-2:] == 'er':
        word = word[:-3]
    elif:
        # lots more cases!
        ...

    return word
```

A. `playing`

B. **`stemming`**

C. `spammer`

D. **`reader`**

E. more than one (which ones?)

How could you fix the ones it gets wrong?

Be Careful!

```
def stem(word):
    if word[-3:] == 'ing':
        if word[-4] == word[-5]:
            word = word[:-4]
        else:
            word = word[:-3]
    elif word[-2:] == 'er':
        word = word[:-3]
    elif:
        # lots more cases!
        ...
    return word
```

```
stem('stemming')  ⇒  'stem'

stem('killing')   ⇒  'kil'

stem('sing')      ⇒  IndexError
```

(original version gives `'s'`)

Things to Consider When Stemming

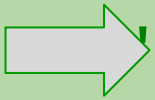
- You could include the length of the word in some rules.
- You could use a dictionary of special cases.
- Be careful about the order in which rules are applied.
- Consider the use of recursion in some cases:

```
stem('readers')
    remove the 's' to get 'reader'
    stem_rest = stem('reader')
        remove 'er' to get 'read'
        return 'read'
```

Things to Consider When Stemming

- You could include the length of the word in some rules.
- You could use a dictionary of special cases.
- Be careful about the order in which rules are applied.
- Consider the use of recursion in some cases:

```
stem('readers')
    remove the 's' to get 'reader'
    stem_rest = stem('reader')
        remove 'er' to get 'read'
        return 'read'
    return stem_rest
```



```
'read'
```

- ***It doesn't need to be perfect!***

There's No "Right Answer"!

- Example: Rather than doing this:

```
stem('party')     ⇒  'parti'
stem('parties')   ⇒  'parti'
```

we could do this instead

```
stem('party')     ⇒  'party'
stem('parties')   ⇒  'party'
```

Classifying a New Body of Text

Suppose we're just focused on the word frequencies:

William Shakespeare
WS: { "love": 50,
"spell": 8,
"thou": 42 }

J.K. Rowling
JKR: { "love": 25,
"spell": 275,
"potter": 700 }

New: { "love": 3, "thou": 1,
"potter": 2 }

How could we give a similarity score for this **new** dictionary against each one above?

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

WS: { **"love": 50**,
"spell": 8,
"thou": 42 }

100 words in all

score vs. WS

$$\frac{\textbf{50}}{\textbf{100}}$$

love

New: { **"love"**: 3, "thou": 1,
"potter": 2 }

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

WS: { **"love": 50**,
"spell": 8,
"thou": 42 }

100 words in all

score vs. WS

$$\frac{50}{100} \cdot \frac{50}{100} \cdot \frac{50}{100}$$

love love love

New: { **"love"**: **3**, "thou": 1,
"potter": 2 }

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

WS: { "love": 50,
"spell": 8,
"thou": 42 }

100 words in all

score vs. WS

$$\frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{\mathbf{42}}{\mathbf{100}}_{\text{thou}}$$

New: { "love": 3, **"thou": 1**,
"potter": 2 }

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

"potter" is not here.

WS: { "love": 50,
"spell": 8,
"thou": 42 }

100 words in all

score vs. WS

$$\frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{42}{100}_{\text{thou}} \cdot \frac{\mathbf{0}}{\mathbf{100}}_{\text{potter}} \cdot \frac{\mathbf{0}}{\mathbf{100}}_{\text{potter}}$$

New: { "love": 3, "thou": 1,
"potter": 2 }

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

"potter" is not here.

WS: { "love": 50,
"spell": 8,
"thou": 42 }

100 words in all

score vs. WS

$$\frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{42}{100}_{\text{thou}} \cdot \frac{\mathbf{0}}{100}_{\text{potter}} \cdot \frac{\mathbf{0}}{100}_{\text{potter}}$$

score = **0**

New: { "love": 3, "thou": 1,
"potter": 2 }

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

"potter" is not here.

WS: { "love": 50,
"spell": 8,
"thou": 42 }

100 words in all

score vs. WS

$$\frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{42}{100}_{\text{thou}} \cdot \frac{\mathbf{0.5}}{100}_{\text{potter}} \cdot \frac{\mathbf{0.5}}{100}_{\text{potter}}$$

score = 0.00000131

New: { "love": 3, "thou": 1,
"potter": 2 }

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

"potter" is not here.

"thou" is not here.

WS: { "love": 50,
"spell": 8,
"thou": 42 }

100 words in all

JKR: { "love": 25,
"spell": 275,
"potter": 700 }

1000 words in all

score vs. WS

$$\frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{50}{100}_{\text{love}} \cdot \frac{42}{100}_{\text{thou}} \cdot \frac{\mathbf{0.5}}{100}_{\text{potter}} \cdot \frac{\mathbf{0.5}}{100}_{\text{potter}}$$

score = 0.00000131

>

score vs. JKR

$$\frac{25}{1000}_{\text{love}} \cdot \frac{25}{1000}_{\text{love}} \cdot \frac{25}{1000}_{\text{love}} \cdot \frac{\mathbf{0.5}}{1000}_{\text{thou}} \cdot \frac{700}{1000}_{\text{potter}} \cdot \frac{700}{1000}_{\text{potter}}$$

score ~= 0.00000000382

New: { "love": 3, "thou": 1,
"potter": 2 }

more likely to be WS!

problem: scores can become too small!

Naïve Bayes Scoring Algorithm

multiply each word's probability as if they were all independent

"potter" is not here.

"thou" is not here.

WS: { "love": 50,
"spell": 8,
"thou": 42 }

100 words in all

JKR: { "love": 25,
"spell": 275,
"potter": 700 }

1000 words in all

score vs. WS

$$\frac{50}{100} \cdot \frac{50}{100} \cdot \frac{50}{100} \cdot \frac{42}{100} \cdot \frac{0.5}{100} \cdot \frac{0.5}{100}$$

love love love thou potter potter

$$3\log\left(\frac{50}{100}\right) + 1\log\left(\frac{42}{100}\right) + 2\log\left(\frac{0.5}{100}\right)$$

score ~= -13.54

score vs. JKR

$$\frac{25}{1000} \cdot \frac{25}{1000} \cdot \frac{25}{1000} \cdot \frac{0.5}{1000} \cdot \frac{700}{1000} \cdot \frac{700}{1000}$$

love love love thou potter potter

$$3\log\left(\frac{25}{1000}\right) + 1\log\left(\frac{0.5}{1000}\right) + 2\log\left(\frac{700}{1000}\right)$$

score ~= -19.38

>

New: { "love": 3, "thou": 1,
"potter": 2 }

more likely to be WS!

problem: scores can become too small!
solution: sum logs!